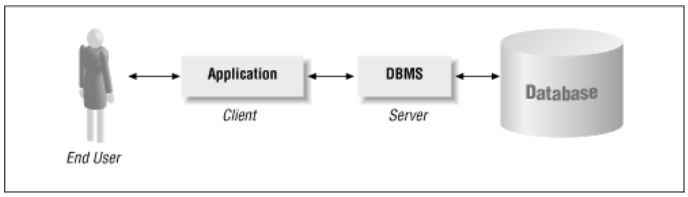
# RICHIAMO DI SISTEMI CENTRALIZZATI

Un **Database Management System *(DBMS)*** è un sistema *(prodotto software)* in grado di gestire delle collezioni di dati che siano:

- **Grandi**: di dimensioni molto maggiori della memoria centrale dei sistemi di calcolo utilizzati.
- **Persistenti**: hanno un periodo di vita indipendente dalle singole esecuzioni dei programmi che le utilizzano.
- **Condivise**: sono dati utilizzati da applicazioni diverse.
- **Affidabili**: i dati devono essere gestiti in modo che siano resistenti ai guasti *(rotture di dischi, cali di corrente, ecc)*. Chiaramente però bisogna garantire anche un'affidabilità rispetto alla sicurezza.

Chiaramente se dobbiamo, per esempio, gestire dei dati di un'applicazione ai quali deve accedere solo una persona e non abbiamo necessità di rendere i dati persistenti non serve utilizzare un DBMS, ma potrebbero bastare delle strutture dati come ad esempio delle hash maps.

Generalmente un DBMS si pone tra la base di dati fisica *(i dischi)* e le applicazioni che accedono ai dati. Ad una applicazione poi possono accedere più persone. Esiste poi anche un **DBA *(Database Administrator)*** che si occupa della manutenzione del DBMS *(sicurezza, controllo degli accessi, ecc)*.




L'**architettura dati** di un DBMS centralizzato è stata definita dall'ente *ANSI/SPARC* ed è caratterizzata da tre livelli:

1. **Schema esterno**: Rappresenta ciò che vede l'utente finale, magari nascondendo alcuni campi o rinominandone altri a seconda dell'utente stesso. Permette quindi agli utenti di avere viste personalizzate dei dati.
2. **Schema logico *(o concettuale)***: Rappresenta la semantica utilizzata per l'utilizzo del DBMS e fa quindi riferimento al modello utilizzato per i dati ed è indipendente dalla tecnologia utilizzata. Rappresenta il design stesso del database a livello concettuale e ne descrive la struttura, rappresenta quali dati sono immagazzinati nel database e le relazioni tra essi.
3. **Schema fisico**: Questo schema rappresenta l'approccio fisico del sistema e quindi le modalità utilizzate da un DBMS per implementare le sue tabelle, le sue configurazioni, tutti i dati di cui ha bisogno sulla memoria fisica stessa.
   Ad esempio una possibilità per implementare una tabella potrebbe essere quella di gestire un unico grande file relativo ad essa.
   Questo livello rappresenta quindi come il DBMS immagazzina i suoi dati, come ottimizza le strutture interne di essi, come li comprime e li cifra.

Le principali caratteristiche di un **DBMS Centralizzato** sono:

- Un **unico schema logico**, ovvero un'unica semantica.
- Un'**unica base di dati**, ovvero un unico insieme di ***record*** interrogati ed aggiornati da tutti gli utenti.
- Nessuna forma di **eterogeneità concettuale** *(non c'è nulla di distribuito)*.
- Un **unico schema fisico**, ovvero un'unica rappresentazione fisica dei dati.
- Nessuna **distribuzione** e nessuna **eterogeneità fisica**.
- Un **unico linguaggio di interrogazione**, quindi un'unica **modalità di accesso** e selezione dei dati di interesse. I DMBS spesso utilizzano un proprio ***dialetto*** di SQL.
- Un **unico sistema di gestione**, quindi un'unica modalità di accesso, aggiornamento e gestione delle transazioni.
- Un'unica **modalità di ripristino** a fronte di guasti.
- Un unico amministratore dei dati e quindi nessuna **autonomia gestionale**.

Come detto prima, le basi di dati sono **grandi** e **persistenti**.
La persistenza richiede una gestione in **memoria secondaria *(di massa)*** visto che la **memoria centrale *(RAM)*** è volatile. La grandezza richiede che tale gestione sia sofisticata visto che non possiamo caricare tutto in memoria principale.
Le principali azioni nei DBMS riguardano quindi la memorizzazione fisica efficienti delle strutture dati nella memoria secondaria e la scelta efficiente delle pagine da trasferire nel buffer della memoria centrale.

Seppur sia vero che al giorno d'oggi i livelli prestazionali delle memorie secondarie siano molto migliori *(i tempi di accesso alle memorie flash degli SSD iniziano ad essere comparabili con i tempi di accesso alla memoria centrale o comunque sicuramente migliori dei tempi degli HDD)*, è anche vero che in termini di affidabilità le tecnologie SSD sono ancora in svantaggio rispetto ai dischi magnetici.

Per poter ovviare al problema delle dimensioni si utilizza quindi un **buffer** di memoria che solitamente opera secondo un principio di località per cui se accediamo ad una certa riga è probabile che si voglia accedere anche a quelle vicine. Inoltre, si utilizzano anche delle strutture dati efficienti per l'accesso a questi dati; se ad esempio abbiamo un file molto grande relativo ai dati, un accesso sequenziale può essere poco efficiente quando ci interessano solo le sue ultime righe. Vedremo che va prestata particolare attenzione alle operazioni di scrittura sul buffer quando si tratta di DBMS.

Una base di dati è una risorsa **integrata** e **condivisa** fra varie applicazioni. Questo chiaramente pone dei problemi di sicurezza e rende necessari dei **meccanismi di autorizzazione** e dei problemi di **concorrenza** che rendono necessario un **controllo della concorrenza**. In un mondo ideale, le transazioni sono corrette se sono **seriali**, ma questo penalizzerebbe di molto l'efficienza del sistema e quindi il controllo della concorrenza permette un ragionevole compromesso. Inoltre, il controllo della concorrenza permette anche di garantire la **consistenza** del sistema, ma a volte potrebbe essere più conveniente prediligere la concorrenza piuttosto che la consistenza.

Le basi di dati vengono interrogate: gli utenti vedono il modello logico *(relazionale)* mentre i dati sono in memoria secondaria. Le strutture logiche tuttavia non sono efficienti in memoria secondaria e servono dunque strutture fisiche opportune. Come detto prima inoltre la memoria secondaria è molto più lenta della memoria principale e serve dunque un'interazione fra memoria principale e secondaria che limiti il più possibile gli accessi alla secondaria. È necessario quindi **ottimizzare** le **esecuzioni delle interrogazioni** *(ad esempio quando abbiamo delle chiamate con delle join che rischia di produrre una tabella risultante di dimensioni $m \times n$)*.

Le basi di dati sono **affidabili** e devono quindi essere conservate anche in presenza di malfunzionamenti. Pensiamo ad un trasferimento di fondi da un conto corrente ad un altro con un guasto a metà. Per permettere questa affidabilità le transazioni devono essere **atomiche** *(o vengono eseguite tutte le azioni di una transazione o nessuna)* e **definitive** *(dopo la conclusione, non si dimenticano gli effetti)*.

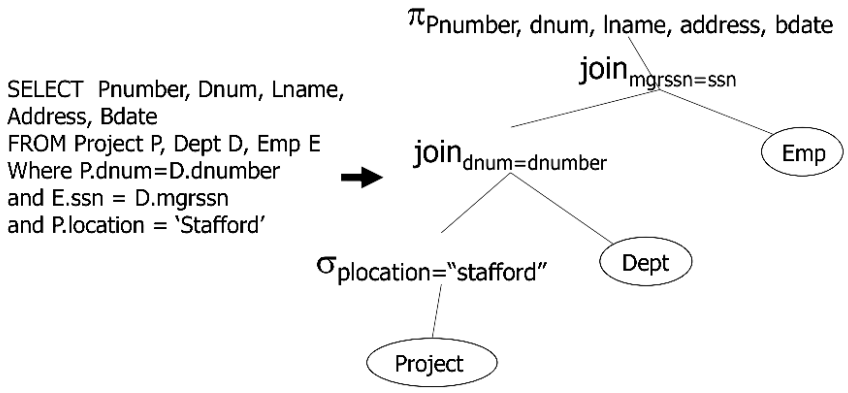
Per garantire tutte queste funzionalità un DBMS è costituito da varie parti che cooperano come un *gestore di interrogazioni* che ottimizza le query e le passa al *gestore dei metodi d'accesso* che le trasforma in comandi di accesso a pagine e li invia al *gestore del buffer* che ottimizza la gestione del buffer e invia i comandi al *gestore della memoria secondaria* che le traduce in accesso a pagine sul disco. Il tutto deve avvenire in pochissimo tempo.

## OTTIMIZZAZIONE DELLE INTERROGAZIONI

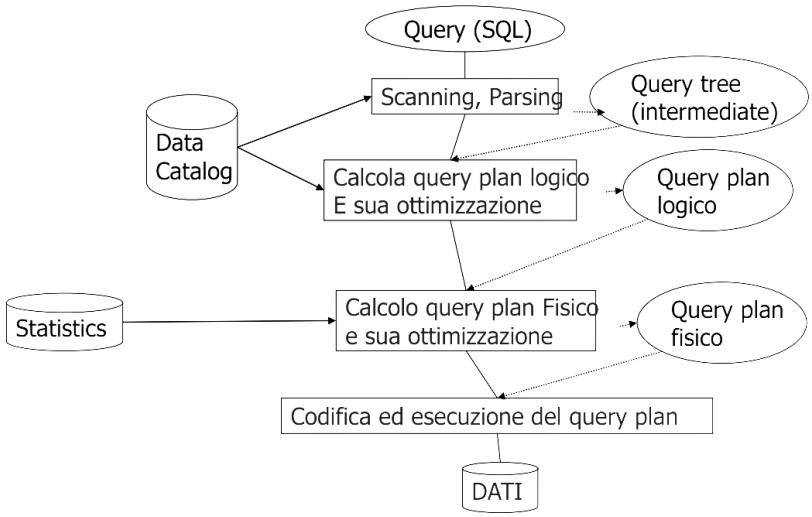
Per fare l'ottimizzazione delle interrogazioni vengono eseguiti diversi passaggi.
Chiaramente il primo passaggio consiste nel verificare che la query sia sintatticamente corretta *(scanning e parsing)*. Per controllare che tutti i nomi delle tabelle siano effettivamente relativi a delle tabelle esistenti si utilizza il **Data Catalog**, un database particolare presente in qualsiasi DBMS, che contiene informazioni sugli altri database del DBMS *(tutte le tabelle e tutti gli attributi)*. Dopo questa operazione si costruisce un **query tree**. In seguito viene calcolato un **query plan** *(che permette di ottimizzare la query andando ad esempio ad eseguire le select prima delle join)*. Il query plan contiene operazioni di algebra relazionale sulle tabelle logiche dello schema.
Un altro database particolare è **Statistics** contenente statistiche sulla storia di esecuzioni precedenti delle interrogazioni e permette di ottimizzare quindi maggiormente il query plan.
Infine, il query plan logico viene tradotto in un query plan fisico.

Chiaramente queste trasformazioni vengono eseguire mediante una strategia che usa **proprietà algebriche** e una stima dei costi delle operazioni fondamentali per i diversi metodi di accesso. In generale il problema di ottimizzazione delle query è esponenziale, ma si utilizzano delle approssimazioni ragionevoli basate su euristiche.

Possiamo osservare un esempio di trasformazione di una query in query plan logico. Notiamo come venga eseguita prima la selezione.




Osserviamo anche lo schema completo dell'ottimizzazione delle query.




# TRANSAZIONI

Le transazioni sono degli insiemi di istruzioni di lettura e scrittura sulla base di dati che godono di alcune proprietà che permettono la loro corretta esecuzioni in ambienti concorrenti e non affidabili. Generalmente vengono identificate da un inizio **begin - transaction** *(*`start transaction` *in SQL)* e una fine **end - transaction** *(non specificata in SQL)* e al cui interno deve essere eseguito una e una sola volta uno dei seguenti comandi:

- **commit work** per terminare correttamente
- **rollback work** per abortire la transazione

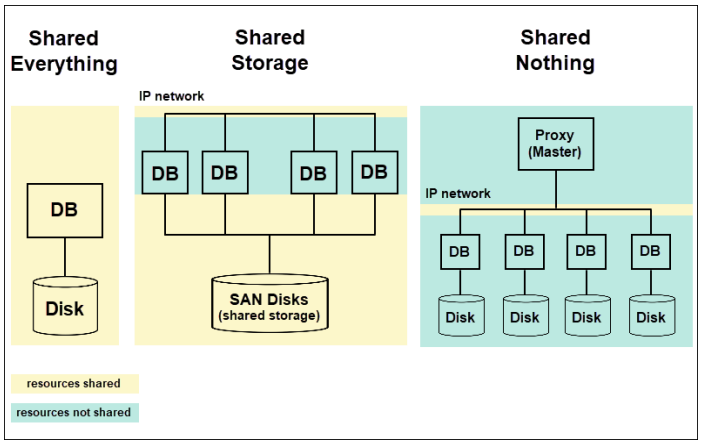
Un **sistema transazionale *(OLTP)*** è in grado di definire ed eseguire transazioni per un certo numero di applicazioni concorrenti.

Le transazioni godono delle proprietà *ACID (Atomicità, Consistenza, Isolamento, Durata (persistenza))*. In particolare, l'isolamento fa si che ogni transazione venga eseguita a sé stante, senza cioè avere accesso agli stati intermedi delle altre. Si dice inoltre che un insieme di transazioni eseguire concorrentemente sono isoalte se producono lo stesso risultato di una *(qualsiasi)* esecuzione sequenziale di esse.

# BASI DI DATI DISTRIBUITE ETEROGENEE ED AUTONOME *(DEA)*

Abbiamo visto come un'architettura centralizzata preveda tante postazioni di lavoro che accedono a delle applicazioni che a loro volta comunicano con un unico DBMS collegato ad una o più basi di dati.

Nel caso di sistemi distribuiti possiamo avere diverse possibilità di architettura.
Un esempio di queste architetture sono le cosiddette **architetture shared nothing**, dove esistono vari ***nodi elaborativi*** ciascuno con la propria base di dati e dove gli utenti accedono alle applicazioni dei nodi elaborativi. I DBMS dei vari nodi elaborativi sono autonomi *(potrebbero anche essere di vendor diversi)* e, se gestiti correttamente, possono essere estesi di molto ***(scalabilità orizzontale)***. Altre architetture sono la *shared everything* dove DBMS e disco sono comuni e la *shared storage* dove esistono diversi DBMS che però usano lo stesso storage fisico, portando quindi a vari problemi di concorrenza.




Una base di dati distribuita può esistere anche all'interno di un'unica organizzazione; pensiamo ad un'azienda che ha delle sedi a Roma, Milano e a Padova. Immaginiamo che inizialmente abbia un'unica base di dati centralizzata a Milano acceduta da tutti i nodi con transazioni centralizzate.

In questa soluzione iniziale il nodo di Milano ha sia la capacità di **data processing** che di **application processing**, mentre i nodi di Roma e di Padova hanno solo capacità di application processing.

Se volessimo avere i dati *vicini a dove servono* potremmo portare parte dei dati a Roma e parte a Padova, creando dei **frammenti** di dati; a questo punto avremmo un sistema dove tutti e tre i nodi hanno sia capacità di data processing che di application processing.
La cosa importante è che in questo modo ogni base di dati è unica rispetto al nodo di cui fa parte; ad esempio se a Padova non ci fossero punti vendita ma solo dipendenti, in quel frammento di dati possiamo rappresentare solo questi ultimi. In più, possiamo pensare di fare delle repliche dei dati; possiamo avere ad esempio nel nodo di Milano anche dati relativi al nodo di Roma perché magari il nodo di Milano ha bisogno di accedere anche a dati di Roma.

Detto ciò, avendo una base di dati distribuita ci si può anche porre il problema dell'avere più di un database fisico. Chiaramente avendo più database abbiamo anche un'eterogeneità maggiore. Inoltre, essendo sistemi diversi c'è anche un'autonomia alta che si rispecchia in una difficoltà maggiore nel farli collaborare.

| SISTEMA | DISTRIBUZIONE | ETEROGENEITÀ | AUTONOMIA |
|---|---|---|---|
| Base di dati distribuita | Alta | Bassa | Bassa |
| Multi Database | Alta | Alta | Alta |

Nella realizzazione di un sistema con caratteristiche di distribuzione il primo problema da porsi è **cosa distribuire**; possiamo distribuire le applicazioni *(applicazioni su più nodi elaborativi, elaborazione distribuita)* e i dati *(base di dati distribuita)* e abbiamo chiaramente vari livelli di possibilità.
Pensiamo al paradigma client server: possiamo avere i *terminali "stupidi" (vedono i dati e basta, non hanno capacità di elaborazione)* così come possiamo avere una elaborazione intermedia tra client e server *(magari il client valida i dati o li trasforma)* e una elaborazione totale sul client, lasciando al server il solo compito di gestione dei dati *(il server ha solo i dati, nessuna logica)*.
Solitamente questo passaggio è utile farlo nella fase di progettazione del sistema, ma a volte è necessario considerare il sistema esistente o unire due sistemi informativi diversi *(quando un'organizzazione compra un'altra)*.

Un DBMS Distribuito Eterogeneo Autonomo è in generale una *federazione* di DBMS che collaborano nel fornire servizi di accesso ai dati con livelli di **trasparenza** definiti, dove con *trasparenza* si intende la proprietà di nascondere le diversità tra basi di dati nei nodi del sistema rispetto alla distribuzione, eterogeneità ed autonomia.
L'esigenza di ***integrare a posteriori*** o di ***progettare fin dall'inizio*** sistemi di tipo federato, oltre ad essere stimolata dallo sviluppo delle reti, emerge in diversi casi:

- Evoluzione o integrazione di *componenti applicativi* sviluppati separatamente *(per ragioni tecniche, organizzative o temporali)*
- Cooperazione di *processi* in precedenza separati
- Cooperazione *(o fusione)* di *enti* o *aziende* indipendenti

È possibile dividere i livelli di federazione su tre **caratteristiche ortogonali *(indipendenti)***:

- **Autonomia**: fa riferimento a *quanto* un nodo è autonomo rispetto agli altri.
  Esistono diverse forme di autonomia:
  - **Di progetto**: ogni nodo adotta un proprio modello dati e sistema di gestione delle transazioni.
  - **Di condivisione**: ogni nodo sceglie la porzione di dati che intende condividere con gli altri nodi *(in questo caso lo schema dei dati è unico)*.
  - **Di esecuzione**: ogni nodo decide in che modo eseguire le transazioni che gli vengono sottoposte.

  Abbiamo quindi diversi tipi di DBMS autonomi:
  - **DBMS Strettamente integrati**: in questo caso non c'è nessuna autonomia, i dati sono *logicamente* centralizzati ed esiste un unico data manager responsabile delle transazioni applicative. I data manager locali non operano in maniera autonoma.
  - **Semi - autonomi**: ogni data manager è autonomo ma partecipa alle transazioni globali. Una parte dei dati è condivisa e richiedono modifiche architetturali per poter far parte della federazione.
  - **Totalmente autonomi *(peer - to - peer)***: ogni DBMS lavora in completa autonomia ed è inconsapevole dell'esistenza degli altri.
- **Distribuzione**: fa riferimento alla distribuzione dei dati.
  Possiamo distinguere:
  - **Distribuzione client / server**: la gestione dei dati è concentrata nei server mentre i client forniscono l'ambiente applicativo e la presentazione.
  - **Distribuzione peer - to - peer**: non c'è distinzione tra client e server e tutti i nodi del sistema hanno identiche funzionalità DBMS.
  - **Nessuna distribuzione**: i dati sono centralizzati.
- **Eterogeneità**: rappresenta la diversità del sistema e può riguardare:

    - **Modello dei dati**: Relazionale piuttosto che XML, Object Oriented, JSON, ecc.
    - **Linguaggio di query**: Dialetti SQL, query by example, linguaggi object oriented.
    - **Gestione delle transazioni**: diversi protocolli per il concurrency control e per il recovery.
    - **Schema concettuale e schema logico**: ad esempio un concetto potrebbe essere rappresentato in uno schema come attributo e in un altro come entità.

Sviluppando una o più di queste caratteristiche possiamo quindi avere varie tipologie di DBMS, le più rilevanti sono:

- **DBMS Distribuiti Omogenei *(DDBMS)***: C'è distribuzione ma nessuna eterogeneità e nessuna autonomia. Vedremo specificatamente come vengono distribuiti i dati.
- **DBMS Distribuito Eterogeneo**: In questa fase vederemo il problema dell'integrazione dei dati.
- **Multi Database MS**: Sistemi totalmente autonomi.

# DDBMS

Vedremo per la distribuzione dei dati l'**architettura dei dati** e l'**architettura funzionale**, ovvero l'insieme delle tecnologie a supporto dell'archiettura dati.

Abbiamo visto in precedenza come esistano vari schemi in un architettura dati: dovendo distribuire i dati mantenendo lo schema, dobbiamo introdurre, tra lo schema esterno e lo schema logico, uno **schema logico globale** a cui faranno riferimento i vari *schemi logici locali (che sono delle viste di quello globale)* e gli schemi fisici locali.

Vedremo che le interrogazioni vengono fatte allo schema logico globale che poi le instraderà ai giusti nodi e metterà insieme i risultati. Inoltre, per ogni funzione del DBMS *(query processing, transaction manager, ecc)* vi può essere una gestione **centralizzata / gerarchica** o **distribuita** con ruoli **statici** o **dinamici**.

Chiaramente questo introduce anche una fase di **progettazione della distribuzione** nella progettazione di applicazioni DBMS. Non solo, questa distribuzione introduce anche una **portabilità**, cioè è necessario poter eseguire le stesse applicazioni DB su ambienti runtime diversi *(che magari dipendono da diversi standard SQL)*. Inoltre, abbiamo anche un concetto di **interoperabilità**, ovvero la capacità di eseguire applicazioni che coinvolgono contemporaneamente sistemi diversi ed eterogenei *(di solito per fare ciò vengono usati dei middleware come ODBC, JDBC, ecc ma esistono anche dei protocolli per fornire alcune connettività standard)*.

## VANTAGGI DDBMS

Alcuni dei vantaggi dei DBMS distribuiti sono:

- **Località**: i dati sono *vicino* alle applicazioni che li utilizzano più frequentemente. Riduce i tempi di latenza, i tempi di accesso. Il paradigma è spostare i dati verso le applicazioni, le partizioni dei dati corrispondono spesso a delle partizioni naturali delle applicazioni e degli utenti. La distribuzioni dei dati spesso è flessibile: è possibile spostare un intera tabella così come è possibile spostarne solo un sottoinsieme o replicarla.
- **Modularità**: le modifiche alle applicazioni e ai dati possono essere effettuate a basso costo. Scalabilità orizzontale.
- **Resistenza ai guasti** grazie alla possibilità di ridondanza.
- **Prestazioni / Efficienza**: distribuendo un DB su nodi diversi, ogni nodo gestisce un DB di dimensioni ridotte. Questo significa che i singoli DB sono più facili da gestire e ottimizzare localmente e, in particolare, ogni nodo può adottare delle ottimizzazioni personalizzate. Il carico

inoltre viene distribuito sui nodi. Tutto ciò però richiede chiaramente un coordinamento tra i nodi e aumenta il traffico di rete che può rivelarsi un collo di bottiglia per le prestazioni.

## FUNZIONALITÀ DDBMS RISPETTO AI DBMS CENTRALIZZATI

Come abbiamo detto, nell'architettura shared nothing ogni server gestisce il proprio DBMS e quindi ha una buona capacità di gestire le proprie applicazioni poiché risponde solo a query specifiche di quel DBMS o a transazioni distribuite che richiedono carichi supplementari sul sistema *(se eseguiamo delle scritture su diversi DBMS dobbiamo assicurarci che abbiano avuto successo su tutti i DBMS)*.

Il traffico di rete generato in queste configurazioni è dovuto alle query delle applicazioni e ai relativi risultati del server, dalle richieste transazionali delle applicazioni e dai dati di controllo per il coordinamento dei nodi in caso di transazioni distribuite.

Chiaramente l'elemento critico di questa struttura è la **rete**, dunque si cerca di ottimizzare il sistema per far sì che i nodi siano distribuiti in modo che la maggior parte delle transazioni siano locali.

Dunque un DDBMS trasmette dati sia relativi a query/frammenti di DB che ai dati di controllo tra i nodi ed esistono diversi livelli di trasparenza rispetto al programmatore di questa frammentazione dei dati *(ci sono soluzioni software che nascondo totalmente la frammentazione dei dati)*. Chiaramente, per fare ciò, il *query processor* deve prevedere un query plan globale che definisca una risoluzione della query tenendo conto di questa frammentazione: deve quindi interrogare lo schema logico globale e poi capire come accedere ai vari dati secondo gli schemi logici locali *(questo accesso può anche essere demandato ai nodi locali)*.
Chiaramente ci deve essere anche un controllo di concorrenza e delle strategie di recovery sia locali che globali.

## FRAMMENTAZIONE

Immaginiamo di avere una tabella relazionale *(vediamola come una matrice con righe e colonne)*.
Esistono due tipi di frammentazione:

1. **Frammentazione orizzontale**: distribuisco le righe della tabella. Lo schema rimane inalterato per tutti i frammenti e otteniamo delle sotto - tabelle più piccole.

   Notiamo che i frammenti non sono altro che delle selezioni della tabella originale.
   Possiamo ricostruire la tabella originale facendo due selezioni e unendo i risultati.

EMPLOYEE1

| EmpNum | Name | DeptName | Salary | Tax |
|---|---|---|---|---|
| 1 | Robert | Production | 3.7 | 1.2 |
| 2 | Greg | Administration | 3.5 | 1.1 |
| 3 | Anne | Production | 5.3 | 2.1 |

EMPLOYEE2

| EmpNum | Name | DeptName | Salary | Tax |
|---|---|---|---|---|
| 4 | Charles | Marketing | 3.5 | 1.1 |
| 5 | Alfred | Administration | 3.7 | 1.2 |
| 6 | Paolo | Planning | 8.3 | 3.5 |
| 7 | George | Marketing | 4.2 | 1.4 |

2. **Frammentazione verticale**: distribuisco le colonne. Chiaramente in questo caso ogni frammento avrà un proprio schema che corrisponde ad uno schema ridotto rispetto allo schema originale.

**NOTA**: pur essendo schemi indipendenti, tutti i vari sotto - schemi condivideranno un attributo in comune che quindi verrà replicato in tutti che corrisponde proprio alla **chiave primaria** della tabella. Questo viene fatto poiché, per garantire la trasparenza, è necessario anche poter tornare indietro dalla frammentazione ad un'unica tabella tramite un join. Chiaramente una eventuale modifica alla chiave primaria va propagata su tutti i frammenti per mantenere la ricostruibilità.

Notiamo che i frammenti sono delle proiezioni della tabella originale.

EMPLOYEE1

| EmpNum | Name |
|---|---|
| 1 | Robert |
| 2 | Greg |
| 3 | Anne |
| 4 | Charles |
| 5 | Alfred |
| 6 | Paolo |
| 7 | George |

EMPLOYEE2

| EmpNum | DeptName | Salary | Tax |
|---|---|---|---|
| 1 | Production | 3.7 | 1.2 |
| 2 | Administration | 3.5 | 1.1 |
| 3 | Production | 5.3 | 2.1 |
| 4 | Marketing | 3.5 | 1.1 |
| 5 | Administration | 3.7 | 1.2 |
| 6 | Planning | 8.3 | 3.5 |
| 7 | Marketing | 4.2 | 1.4 |

Le frammentazioni devono rispettare delle regole di correttezza quali:

- **Completezza**: ogni record della relazione $R$ di partenza deve poter essere ritrovato in almeno uno dei frammenti.
- **Ricostruibilità**: la relazione $R$ di partenza deve poter essere ricostruita senza perdita di informazione a partire dai frammenti.
- **Disgiunzione**: ogni record della relazione $R$ deve essere rappresentato in uno solo dei frammenti *(o su una tabella o su un insieme di tabelle)* o in alternativa **replicazione**.

Chiaramente in caso di frammentazione lo ***schema globale*** esiste solo virtualmente, quello che esiste realmente è un **catalogo**, ovvero un *mapping*, una tabella che tiene traccia dei vari frammenti e su quali nodi sono allocati.

## REPLICAZIONE

Vediamo gli aspetti positivi della replicazione:

- Migliora le prestazioni
  Consente la coesistenza di applicazioni con requisiti operazionali diversi sugli stessi dati e ne aumenta la località. Questo è particolarmente utile quando gli accessi sono solo in lettura.

Ci sono però anche degli aspetti negativi:

- Introduce complicazioni architetturali
  Specialmente in caso di scritture che vanno propagate in tutte le repliche
- Richiede un nuovo passo di progettazione in cui si stabilisce quali frammenti replicare, quante copie mantenere, dove allocarle e la politica di gestione delle copie.

## TRASPARENZA

La trasparenza è importante perché aiuta a separare la logica applicativa dalla logica dei dati ma richiede uno strato software che gestisca la traduzione dallo schema unico ai vari sotto - schemi il che porta ad un aumento della complessità del sistema e ad un calo di prestazioni. Tuttavia se il mapping è nativamente supportato dal DDBMS i problemi si riducono.

Esistono due tipi di trasparenze:

1. **Trasparenza logica**: indipendenza dell'applicazione da modifiche dello schema *logico*.
   Un'applicazione che utilizza un frammento dello schema non subisce modifiche quando altri frammenti vengono modificati.

2. **Trasparenza fisica**: indipendenza dell'applicazione da modifiche allo schema *fisico*.

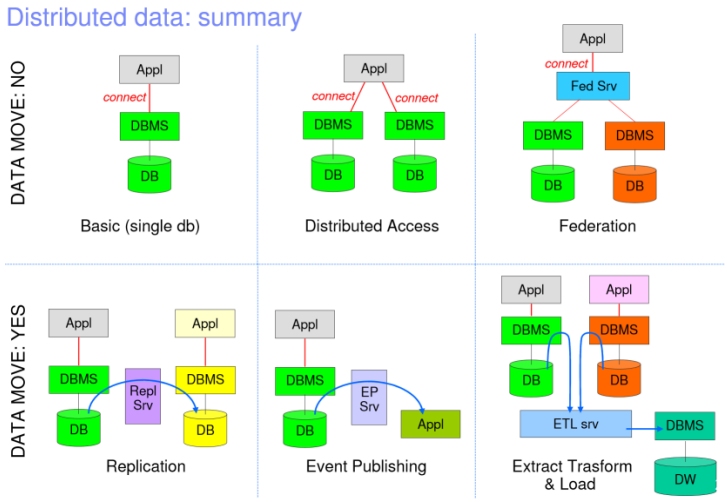
Inoltre esistono tre livelli di trasparenza:

1. **Trasparenza alla frammentazione**: ignora l'esistenza dei frammenti.
   La traduzione della query globale in quelle locali è a carico del sistema. Chiaramente va fatto un *query - rewriting* per scomporre una query globale in più query locali.
2. **Trasparenza all'allocazione**: il programmatore sa che esistono dei frammenti ma ignora la loro allocazione. Le query locali sono scritte direttamente dal programmatore.
3. **Trasparenza al linguaggio**: livello minimo, l'applicazione deve specificare sia i frammenti che il loro nodo. Un nodo può offrire interfacce che non sono standard SQL ma l'applicazione viene scritta a prescidere in SQL standard.

## TIPI DI DBMS

Abbiamo visto che esistono diverse architetture di DBMS come quelle in cui esiste un solo DB, le shared nothing dove esistono più DB, le federazioni dove i DB sono anche di vendor diversi che usan un *federation server* per interfacciarsi coi DBMS. In queste prime tre architetture tuttavia i dati rimangono fermi.

Abbiamo anche architetture che spostano i dati come le architetture di replica dove abbiamo un componente che si occupa di mantenere aggiornate le repliche, architetture basate su event publishing o architetture in cui più applicazioni coi propri database collaborano.




## STRATEGIE DI QUERY PROCESSING

Vedremo che esistono diverse strategie di query processing in base al tipo di operazione fatta; ad esempio se l'operazione è una read non avremmo problemi di consistenza dei dati ma solo di disponibilità quindi si possono usare anche strategie di replicazione. Quando andiamo a scrivere invece bisogna considerare la consistenza dei dati e quindi, in caso di repliche, bisogna pensare ad una strategia di aggiornamento delle repliche.

Abbiamo visto che per fare il processing delle query esistono vari passaggi:

- **Query Decomposition**: questa fase opera sullo schema logico globale, non tiene conto della distribuzione dei dati e utilizza tecniche di ottimizzazione algebrica analoghe a quelle delle soluzioni centralizzate.

  Produce in output un query tree non ottimizzato rispetto ai costi di comunicazione.

- **Data Localization**: a questo punto viene considerata la distribuzione dei frammenti.
  Vengono ottimizzate le operazioni rispetto alla frammentazione con **tecniche di riduzione**.
  Ad esempio, se abbiamo una query, mentre nel caso di frammentazione orizzontale siamo obbligati a farla su tutti i frammenti *(a meno che non abbiamo informazioni aggiuntive, ad esempio se sapiamo che sul nodo 1 ci sono le chiavi primarie da 1 a 500 e dobbiamo selezionare la chiave primaria 350)*, nel caso di frammentazione verticale dobbiamo farla su tutti solo se stiamo considerando la chiave primaria.

  Produce in output una query che opera in modo efficiente sui frammenti ma che non è ancora ottimizzata.
- **Global Query Optimization**: a questo punto si può pensare ad una ottimizzazione globale basata su delle *statistiche dei frammenti*. Si arricchisce quindi l'albero di query con le operazioni di *send* e *receive*.
  Chiaramente a questo punto si può pensare anche di parallelizzare alcune operazioni indipendenti.
  L'obiettivo è trovare l'ordinamento *migliore* delle operazioni definite dalla fragment query tenendo conto anche dei costi di comunicazione.

  Produce in output le decisioni più importanti riguardanti i ***join*** poiché è l'operatore più complicato da eseguire dato che selezione e proiezione riducono il volume di dati da trattare, mentre il join lo aumenta e richiede un controllo per tutte le coppie di valori delle tabelle.
  Vedremo poi che in questa fase si sceglie anche l'ordine dei join *n*-ari e se eseguire un join o un semi - join.

  Chiaramente la scelta dell'ottimizzazione dipende da diversi fattori come la velocità della rete tra un nodo e gli altri, il carico applicativo, la capacità computazionale dei nodi. Ad esempio se dobbiamo fare una selezione e un join, sapendo di avere una connessione scarsa con i nodi ma un buon potere computazionale su tutti i nodi, ha più senso eseguire i join frammentati sui vari nodi e poi inviare il risultato al nodo finale che farà l'unione dei risultati frammentati. Viceversa se abbiamo una buona connessione e magari solo il nodo finale ha una buona capacità computazionale potrebbe avere più senso fare solo le selezioni sui nodi frammentari ed inviare quei risultati in modo che poi il join venga fatto dal nodo finale. In questo secondo caso però non si potranno utilizzare strategie come l'hash join poiché gli indici non sono portabili da una macchina all'altra, di conseguenza il join potrebbe risultare meno performante.

  Generalmente i tempi di esecuzione risultano trascurabili rispetto a quelli di trasferimento ma non è sempre detto e dipende molto dalla topologia e dal carico applicativo.
- **Local Query Optimization**: ogni nodo riceve una fragment query e la ottimizza in modo indipendente usando tecniche analoghe a quelle dei sistemi centralizzati.

Rispetto al caso centralizzato dove i costi più rilevanti sono quelli di trasferimento dei blocchi da memoria secondaria a principale, qui consideriamo solo i costi di comunicazione e trascuriamo quelli di I/O.
Avremo quindi $costo\ comunicazione = C_{MSG} * \#msg + C_{TR} * \#bytes$ dove $C_{MSG}$ è il costo fisso di spedizione/ricezione dei messaggi e $C_{TR}$ è il costo *(fisso rispetto alla topologia)* di trasmissione dei dati.

Solitamente nelle reti geografiche i costi di comunicazione sono molto maggiori dei costi di I/O, mentre nelle reti locali sono comparabili.

Possiamo essere interessati a

- Minimizzare il tempo di risposta: introducendo più parallelismo, portando però ad un aumento del costo totale.

- Minimizzare il costo totale: non tenendo conto del parallelismo, aumenta il throughput aumentando il response time.

### JOIN E SEMI - JOIN

Supponiamo di dover fare un join tra due tabelle su due nodi diversi.
In alcune circostanze il semi - join può risultare un'alternativa più efficiente al join.

Il semi - join è $R\ semi-join_A\ S = \pi_{R^*}(R\ join_A\ S)$
Dove $R^*$ è l'insieme degli attributi di $R$.
Questo vuol dire che, invece importare tutta la tabella $S$ nel nodo di $R$, importiamo solo $S.A$, ovvero solo l'attributo sul quale facciamo il join.

Il semi - join $R\ semi-join_A\ S$ è quindi la proiezioni sugli attributi di $R$ dell'operazione di join.
**NOTA**: il semi - join non è comutativo.

Chiaramente l'uso del semi - join è conveniente se il costo del suo calcolo e del trasferimento del risultato sono inferiori al costo di trasferimento dell'intera relazione del costo del join intero.

Abbiamo dunque visto le basi di dati DEA con casi di DDBMS e Multidatabase *(più DBMS utilizzati dallo stesso insieme di applicazioni)*. Vedremo poi le basi di dati parallele che portano un incremento prestazionale mediante parallelismo e le basi di dati replicate utili per il recovery.

## CONTROLLO DI CONCORRENZA

Andiamo ora a vedere come operano le query in scrittura che è chiaramente un processo più complicato.
Possiamo avere, dirette ad un unico server remoto:

- **Remote requests**: transazioni *read - only* con un numero arbitrario di query SQL.
- **Remote transactions**: transazioni *read - write* con un numero arbitrario di operazioni SQL *(select, insert, delete, update)*.

Dirette invece ad un numero arbitrario di server possiamo avere:

- **Distributed requests**: *read - only* arbitrarie nelle quali ogni singola operazione SQL si può riferire a qualunque insieme dei server. Richiede un ottimizzatore distribuito.
- **Distributed transactions**: numero arbitrario di operazioni SQL, ogni operazione è diretta ad un unico server. Le transazione possono modificare più di un DB. Richiede un protocollo transazionale di coordinamento distribuito ***(two - phase commit)***. Chiaramente anche qui deve valere la proprietà di atomicità.

La distribuzione non ha conseguenze su consistenza e durabilità:

- **Consistenza**: non dipende dalla distribuzione poiché i vincoli descrivono solo proprietà logiche dello schema *(indipendenti dall'allocazione)*.
- **Durabilità**: garantita localmente da ogni sistema.

È invece necessario rivedere alcuni componenti dell'architettura in merito a

- **Isolamento** tramite concurrency control.
- **Atomicità** tramite reliability control e recovery manager.

## CONCURRENCY CONTROL

L'idea alla base del controllo di concorrenza è che ogni transazione $t_i$ possa essere suddivisa in sotto - transazioni $t_{ij}$ che saranno eseguite sul nodo $j$.

Ad esempio

$t_1 : r_{11}(x)w_{11}(x) \quad r_{12}(y)w_{12}(y)$
$t_2 : r_{22}(y)w_{22}(y) \quad r_{21}(x)w_{21}(x)$

Abbiamo la transazione $t_1$ che fa una lettura e una scrittura prima sulla risorsa $x$ del nodo $1$ e poi sulla risorsa $y$ del nodo $2$ e la transazione $t_2$ che fa una lettura e una scrittura prima sulla risorsa $y$ del nodo $2$ e poi sulla risorsa $x$ del nodo $1$.

Senza alcun controllo di concorrenza il nodo $1$ eseguirà per prima o $r_{11}(x)w_{11}x$ o $r_{21}(x)w_{21}(x)$ in base a quale transazione riceve prima *(e viceversa il nodo 2)*. In questo modo lo schedule globale andrà a dipendere dallo schedule locale di ciascun nodo. Inoltre in questo caso, seppur localmente le transazioni sembrino serializzabili, abbiamo un conflitto a livello di schedule globale poiché se la prima transazione parte bloccando la risorsa $x$ sul nodo $1$ e la seconda bloccando la risorsa $y$ sul nodo $2$, poi entrambe le transazioni aspetteranno il rilascio delle risorse opposte.
Chiaramente lo schedule locale però deve dipendere dallo schedule globale, ovvero dall'ordine nel quale si vogliono eseguire le due transazioni $t_1$ e $t_2$.

## ROWA

Se il DB non è replicato e ogni schedule locale è serializzabile allora lo schedule globale è serializzabile se gli ordini di serializzazione sono gli stessi per tutti i nodi, ovvero se il flusso delle transazioni è lo stesso per tutti i nodi.
Con la replicazione le cose cambiano: ipotizziamo di avere un DB su due nodi con la risorsa $x$ replicata nei due nodi e chiamiamo $x_1$ e $x_2$ le due copie.
Abbiamo due transazioni $t_1 : r(x)\ x = x + 5\ w(x)\ commit$ e $t_2 : r(x)\ x = x * 10\ w(x)\ commit$.
Possiamo trascriverle in $t_1 : r_1(x)w_1(x)$ e $t_2 : r_2(x)w_2(x)$.

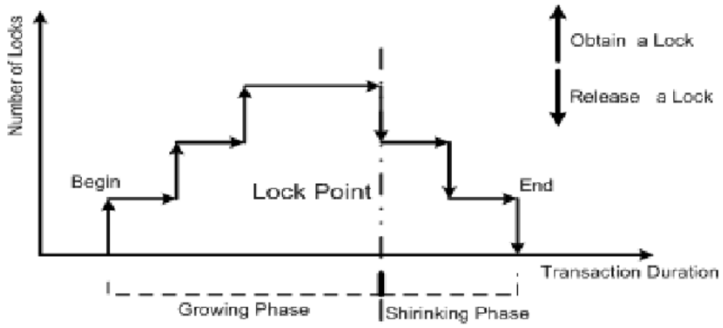
Considerando le seguenti due schedule che possono essere generate ai due nodi $1$ e $2$
$r_{11}(x)w_{11}(x) \quad r_{21}(x)w_{21}(x)$ al nodo $1$
$r_{22}(x)w_{22}(x) \quad r_{12}(x)w_{12}(x)$ al nodo $2$

Si ha che queste sono serializzabili ma serializzano $t_1$ e $t_2$ in ordine opposto come nell'esempio precedente. Viene quindi violata la *mutua consistenza* dei due DB locali per la quale tutte le copie devono avere lo stesso valore al termine della transazione. Abbiamo quindi bisogno di un ***protocollo di controllo delle repliche***.

Un tale protocollo è il **ROWA *(Read Once Write All)*** che, dato un item $x$ *(detto item logico)* con delle repliche $x_1, \ldots, x_n$ *(detti item fisici)*, fa in modo che le transazioni *vedano* solo $x$.
Questo protocollo mappa le $read(x)$ su una qualunque delle copie mentre mappa $write(x)$ su **tutte** le copie. Di fatto, finché non sono accertate tutte le write su tutte le copie, la transazione non continua. Questo protocollo garantisce la consistenza dei dati ma a scapito chiaramente delle performance.
Questa condizione può essere rilassata con dei protocolli asincroni più efficienti ma non ce ne occupiamo.

## TWO - PHASE LOCKING

L'algoritmo two - phase locking prevede che, data una transazione, questa chieda tutti i lock ad essa necessaria e inizia il rilascio di questi solo dopo averli ottenuti tutti. Dunque i lock sono rilasciati solo dopo il commit della transazione.

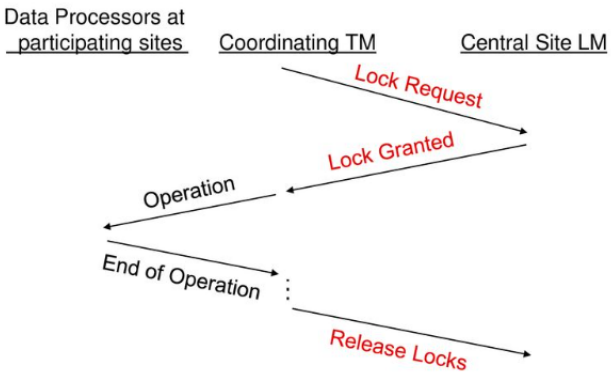



### STRATEGIA CENTRALIZED 2PL

In una strategia centralizzata del 2PL esteso al caso distribuito ogni nodo ha un **lock manager**, tra i quali uno viene eletto ***LM Coordinatore*** il cui compito è gestire i locks per l'intero DDB. Chiaramente ciò implica anche che questo *LM Coordinatore* sia un single point of failure.
Il transaction manager del nodo dove inizia la transazione è considerato ***TM Coordinatore***. La transazione è eseguita anche su altri **Data Processor** e corrispondenti nodi.

### STRATEGIA

1. Il *TM Coordinatore* formula al *LM Coordinatore* le richieste di lock
2. Il *LM Coordinatore* le concede, se possibile, utilizzando un 2PL
3. Il *TM Coordinatore* le comunica ai *DP* in modo che possano continuare a lavorare
4. Alla fine i *DP* communicano al *TM Coordinatore* e il *TM Coordinatore* al *LM Coordinatore* la fine delle operazioni, in modo che il *LM Coordinatore* possa rilasciare i lock.




**PROBLEMA**: come abbiamo detto prima, il nodo dell'unico lock manager può diventare un collo di bottiglia.

### STRATEGIA PRIMARY COPY 2PL

Per evitare il collo di bottiglia visto prima, esiste un'altra strategia di 2PL distribuito.
Questa strategia prevede per ogni risorsa una ***copia primaria*** che viene individuata prima dell'assegnazione dei lock.
Diversi nodi hanno lock managers attivi, ognuno dei quali gestisce una partizione dei lock complessivi relativi alle ***risorse primarie*** **residenti nel nodo**. Per ogni risorsa nella transazione, il *TM Coordinatore* comunica le richieste dei lock al *LM* responsabile della copia primaria che assegna, se possibile, i lock.

Questa strategia evita il bottleneck ma richiede di determinare a priori il lock manager che gestisce ciascuna risorsa ed è necessaria una directory globale in cui tutti vedono tutti per permettere i vari accessi *(ogni nodo deve sapere chi è il lock manager di una determinata risorsa)*. Se vogliamo, qui il collo di bottiglia diventa questa directory globale.

**DEADLOCK DISTRIBUITO**

Chiaramente è possibile che si verifichi un *deadlock distribuito* a causa di attese circolari tra due o più nodi. Questo evento è gestito comunemente nei DDBMS tramite dei time - out.

Esiste anche un algoritmo di rilevazione di deadlock in ambiente distribuito **asincrono** e **distribuito** che può quindi funzionare anche in ambienti *peer - to - peer*. Anche in questo caso indichiamo con $t_{ij}$ la sottotransazione $t_i$ eseguita sul nodo $j$.

Assumiamo che tutte le sotto - transazioni siano attivate in modo asincrono *(tramite Remote Procedure Call bloccante)*: $t_1$ attende che $t_2$ finisca.

Questo può dare luogo a due tipi di attesa:

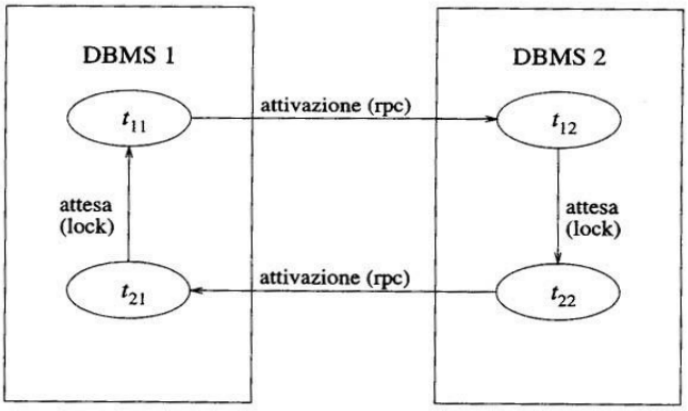
- **Attesa da Remote Procedure Call**
  $t_{11}$ sul nodo $1$ attende $t_{12}$ sul nodo $2$ perché aspetta la sua ***terminazione***.
- **Attesa da Rilascio di Risorsa**
  $t_{11}$ sul nodo $1$ attende $t_{21}$ sullo stesso nodo perché attende il rilascio di una risorsa.

La composizione dei due tipi di attesa può dare luogo a uno stato di deadlock globale.

Possiamo caratterizzare le condizioni di attesa su ciascun nodo tramite delle condizioni di precedenza usando al seguente notazione:

- $EXT_i$ external, chiamata da un nodo remoto $i$
- $x < y$: $x$ attende il rilascio di una risorsa da $y$ *(che può anche essere remoto, ovvero $EXT$)*

La sequenza di attesa generale al nodo $k$ è della forma $EXT < t_{ik} < t_{jk} < EXT$




Chiaramente, da osservatore esterno, riusciamo a vedere quando c'è un ciclo di attese e quindi un deadlock globale, ma a livello di singolo nodo questo è più complicato. Chiaramente, con un architettura *master / slave* il master potrebbe accorgersi di questi cicli, ma diverrebbe un collo di bottiglia.

L'algoritmo distribuito attiva periodicamente sui diversi nodi del DDBMS delle procedure:

1. In ogni nodo, **integra** la sequenza di attesa con le condizioni di attesa locale degli altri nodi logicamente legati da condizioni $EXT$.
2. **Analizza** le condizioni di attesa sul nodo e **rileva** i deadlock locali.
3. **Comunica** le sequenze di attesa ad altre istanze dello stesso algoritmo *(ovvero agli altri nodi)*.

È possibile ovviamente che lo stesso deadlock venga riscoperto più volte; per evitare ciò e rendere più efficiente l'algoritmo si inviano le sequenze di attesa

- **in avanti** verso il nodo ove è attiva la sotto - transazione $t_i$ attesa da $t_j$.
- **solamente quando** $i > j$ dove $i$ e $j$ identificano i nodi.

## RECOVERY MANAGEMENT (ATOMICITÀ)

Un sistema distribuito è soggetto, oltre a guasti locali, anche a perdita di messaggi su rete e partizionamento della rete *(isolamento dei nodi)*.
Possiamo quindi avere vari tipi di guasti:

- **Guasti nei nodi**: possono essere soft o hard, presenti anche nei sistemi centralizzati.
- **Perdita di messaggi**: lasciano l'esecuzione di un protocollo in una situazione di indecisione. Ogni messaggio del protocollo è seguito da un acknowledgment, in caso di perdita del messaggio o dell'ack si genera una situazione di incertezza e non è possibile decidere se il messaggio è stato ricevuto o meno.
- **Partizionamento della rete**: una transazione distribuita può essere attiva contemporaneamente su più sottoreti temporaneamente isolate. Le parti isolate potrebbero quindi prendere decisioni contrastanti sullo svolgimento della transazione.

### THO - PHASE COMMIT

I protocolli di commit consentono ad una transazione di giungere ad una decisione di *abort* / *commit* su ciascuno dei nodi che partecipano ad essa.
Il problema è decidere il *commit* in ambienti distribuiti senza e con presenza di guasti.

**IDEA**: la decisione di *commit* / *abort* tra due o più partecipanti è coordinata e certificata da un ulteriore partecipante.
I server sono chiamati **Resource Manager *(RM)*** ed il coordinatore è chiamato **Transaction Manager *(TM)***.
Il protocollo 2PC si basa sullo scambio di messaggi tra *TM* ed *RM*, i quali mantengono ognuno i propri log.

Possiamo vedere questo protocollo come un'analogia al matrimonio dove il prete chiede ai due partecipanti se sono d'accordo a sposarsi. Allo stesso modo il *TM* chiede a tutti i *RM* se sono d'accordo al *commit*. Se qualcuno risponde negativamente, si effettua un *abort*.

#### IN ASSENZA DI GUASTI

- **PRIMA FASE**
  - Il *TM* chiede a tutti i *RM* come intendono terminare la transazione.
  - Ogni nodo decide autonomamente se fare *commit* o *abort* e comunica unilateralmente la sua decisione **irrevocabile.**
- **SECONDA FASE**
  - Il *TM* prende la decisione globale: se uno solo dei nodi vuole fare la *abort*, è *abort* per tutti, altrimenti *commit.*
  - Il *TM* comunica a tutti la decisione per le azioni locali.

Nei log compaiono due tipi di record

1. **Record di Transazione** contenente informazioni sulle operazioni effettuate.

2. **Record di Sistema** checkpoint *(indica le transazioni attive in quel momento)* e dump *(snapshot del DB)*.

**NOTA**: in genere prima si scrive sul file di log e poi si esegue l'operazione.

Altri record *(TM)*

1. **Record di Prepare** contenente l'identità di tutti i *RM (nodi + transazioni)*.
2. **Global Commit o Abort** che indica come è finita la transazione. La decisione diventa esecutiva quando il *TM* scrive nel proprio log questo record.
3. **Complete** per indicare che la transazione è completata.

Altri record *(RM)*

1. **Ready Record** indica la disponibilità irrevocabile del *RM* a partecipare alla fase di commit.
2. **Not Ready** indica la indisponibilità del *RM* al commit.

In entrambe le fasi possono avvenire dei guasti e in entrambe le fasi i partecipanti devono poter prendere delle decisioni connesse allo stato in cui si trovano. Questo avviene grazie all'utilizzo di timers e stabilendo un intervallo di timeout.

Nella prima fase il *TM* scrive **prepare** nel suo file di log e invia un messaggio **prepare** a tutti i *RM*. A questo punto fissa anche un timeout per indicare il massimo intervallo di tempo di attesa per le risposte.
Gli *RM* che sono recoverable, ovvero pronti a fare il commit, scrivono **ready** nei loro log e inviano un messaggio **ready** al *TM*, mentre quelli che non sono pronti scrivono ed inviano **not-ready** e terminano il protocollo effettuando un *abort*.

A questo punto il *TM* raccoglie tutti i messaggi di risposta.
Se tutti gli *RM* rispondono positivamente, scrive **global commit** nel suo log. Se riceve almeno un **not-ready** o se **scatta il timeout**, scrive **global abort** nel suo log.

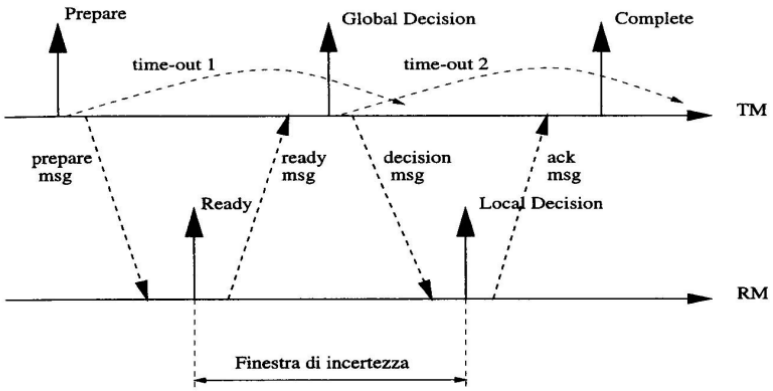
Nella seconda fase il *TM* trasmette la decisione globale e fissa un secondo timeout; gli *RM* che sono ready ricevono il messaggio e scrivono **commit** o **abort** nei propri log e inviano un ack al *TM* ed eseguono le loro azioni locali.
Il *TM* raccoglie tutti i messaggi di ack degli *RM*. Se scatta il timeout, poiché ha già deciso in **modo irrevocabile**, fissa un nuovo timeout e ripete la trasmissione a tutti i *RM* dai quali non ha ancora ricevuto un ack. Quando tutti gli ack sono arrivati, il *TM* scrive **complete** nel suo log.

Chiaramente questo è un paradigma *master / slave* dove la comunicazione avviene solo tra *TM* e ogni *RM*, non c'è nessuna comunicazione diretta tra gli *RM*. Ovviamente il *TM* è un collo di bottiglia.
Una variante alternativa, utile per le reti senza possibilità di broadcast, è una soluzione lineare dove ogni *RM* passa al successivo la decisione del *TM* l'ultimo passa l'ack al *TM*.

Esiste anche una variante **distribuita** dove nella prima fase il *TM* comunica con tutti gli *RM* che poi inviano le loro decisioni a tutti gli altri partecipanti. A questo punto ogni *RM* decide in basi ai voti che *ascolta* dagli altri, eliminando la necessità della seconda fase. Questo però chiaramente genera un grande numero di messaggi in broadcast.

### 2PC CENTRALIZZATO IN CASO DI GUASTI

Un *RM* nello stato ready perde la sua autonomia e aspetta la decisione del *TM*, di conseguenza un guasto nel *TM* lascia gli *RM* in uno stato di incertezza e le risorse allocate alla transazione restano bloccate. L'intervallo tra la scrittura di **ready** nel log dei *RM* e la scrittura di **commit** o **abort** è detta ***finestra di incertezza**.




Vediamo che succede nel caso di guasti di componenti.
Devono essere utilizzati protocolli con **due** diversi compiti:

- Assicurare la terminazione della procedura *(Terminazione)*
- Assicurare il ripristino *(Recovery)*

Supponiamo che cada un *RM*: questo può cadere all'inizio e in questo scattano i timeout e la transazione finisce in abort.
Potrebbe cadere dopo aver dichiarato **abort**: anche qui, la transazione verosimilmente è stata abortita.
Potrebbe cadere dopo aver dichiarato **ready** e  in questo caso può chiedere al *TM* come è finita la transazione *(che lo sa dal proprio file di log)*. Oppure il *TM* potrebbe rimandare la decisione in seguito allo scattare del secondo timeout.

Dunque la caduta di un *RM* non è un problema per questo protocollo.

Diventa più complicato se cade il *TM*.
Può cadere prima della prepare: in questo caso semplicemente gli *RM* possono abortire unilateralmente.
Può cadere dopo la prepare: ipotizzando che il *RM* abbia votato per la commit, attende la decisione globale del *TM*. In questo stato non è in grado di prendere una decisione unilaterale e rimane bloccato in attesa di informazioni. Se scatta il timeout può scegliere di abortire e, se il *TM* si riprende, voterà abort.
Se gli *RM* possono comunicare tra loro possono sbloccare la situazione chiedendo i voti agli altri *RM (algoritmi bizantini)*.

### PROTOCOLLI DI RIPRISTINO

Dopo la caduta del *TM* abbiamo vari casi possibili:

- L'ultimo record del log è **prepare**
  In questo caso il guasto del *TM* può aver bloccato alcuni *RM*.
  Si hanno due opzioni di recovery:
  - Decidere **global abort** e procedere con la seconda fase di 2PC
  - Ripetere la prima fase, sperando di giungere ad un global commit

- L'ultimo record del log è **global-commit** o **global-abort**
  Alcuni *RM* potrebbero non essere stati informati e altri potrebbero essere bloccati, dunque il *TM* ripete la seconda fase fino a ricevere tutti gli ack.
- L'ultimo record del log è **complete**
  In questo caso la caduta del coordinatore non ha effetti.

Dopo la caduta di un *RM* abbiamo vari casi possibili:

- L'ultimo record del log è **abort** o **commit**
  Come nel caso centralizzato, in caso di abort fa l'undo della transazione, in caso di commit la redo.
- L'ultimo record nel log è **Ready**
  In questo caso il *RM* si blocca perché non conosce la decisione del *TM*.
  Dunque può chiedere al *TM* cosa è successo *(richiesta di **remote recovery**)* o rieseguire la seconda fase del protocollo *(aspetta che il TM lo ricontatti)*.

Nel caso di perdita di messaggi o di partizionamento della rete il *TM* non è in grado di distinguere se i timeout scattano per perdita di messaggi o per caduta degli *RM*, ma in entrambi i casi la decisione è un **global abort** in seguito al primo timeout.

Anche durante la seconda fase c'è questa problematica e anche qui la seconda fase viene ripetuta in seguito ai timeout.

Dunque di fatto un partizionamento della rete non causa problemi ulteriori dato che una transazione può avere successo solo se il *TM* e tutti gli *RM* appartengono alla stessa partizione.

#### OTTIMIZZAZIONI

Alcune ottimizzazione del protocollo 2PC comprendono la riduzione del numero di messaggi trasmessi tra coordinatore e partecipanti e la riduzione di scritture nei log.

Ad esempio se un *RM* vede che la propria transazione è *read - only* sa che non influenza l'esito finale della transazione, quindi può rispondere alla prepare con **read - only** e terminare l'esecuzione del protocollo.
In seguito, il *TM* ignora i partecipanti *read - only* nella seconda fase.

Regola **scordarsi gli abort, ricordarsi i commit**
L'idea è che in caso di abort non è necessario scrivere nei log e aspettare la risposta degli *RM*. Dunque se il *TM* riceve una richiesta di remote recovery gli basta controllare che ci sia o meno la commit nei log. Gli unici record scritti sono **global commit**, **ready** e **commit**.

## REPLICA DEI DATI

Vedremo che le tecnologie, le modalità e le architetture di base sia per la distribuzione dei dati che per la loro replica hanno delle caratteristiche comuni indipendentemente dal modello dati su cui si opera.

**DEFINIZIONE**
Il **Database Replication** è il processo che consiste nel creare e mantenere molteplici istanze dello stesso database ed il processo di condividere dati o cambiamenti di design del database tra le varie istanze senza dover copiare l'intero database. Queste repliche possono essere fatte sia in maniera

asincrona che sincrona.
L'elemento principale di questi processi è la **sincronizzazione**, ovvero quella fase che si assicura che, prima o poi, ogni copia del database avrà gli stessi oggetti e gli stessi dati. Può sembrare ovvio volere immediatamente delle copie aggiornate, ma in molti ambiti applicativi questo non per forza è richiesto.

I principali benefici della replicazione sono la disponibilità dei dati, l'affidabilità, la performance, il load balancing, il poter lavorare in ambienti disconnessi e il supporto a molti utenti.

## REPLICHE SINCRONE / ASINCRONE

- **SINCRONE**: gli update dei dati replicati sono parte della fase finale della transazione.
  Se uno o più siti che mantengono una replica non sono disponibili, la transazione non può essere conclusa. Richiedono un grande numero di messaggi per coordinare la sincronizzazione.

  Un esempio di questo tipo di repliche è il protocollo *ROWA*.

  Dunque questa replica obbliga ad aggiornare più database contemporaneamente e permette una grande disponibilità e una perdita di dati minimale, quindi è molto utile nel disaster recovery poiché permette un auto fail - over. Tuttavia, richiede tanto traffico di rete, è poco scalabile, è costosa e meno flessibile. Viene utilizzata nelle applicazioni *mission - critical*. In altre applicazioni potrebbe non essere *cost - efficiently* come soluzione poiché magari i costi sono talmente alti da superare i possibili danni di data loss. Per garantire tutto ciò spesso vengono usati dei backup locali assieme a dei backup geograficamente distanti dai dati primari.
- **ASINCRONE**: in questa modalità i database replicati vengono aggiornati dopo aver aggiornato il database primario. Esiste quindi un *delay* per la riacquisizione della consistenza che può variare da pochi secondi a ore o giorni.

  Questo tipo di replica quindi fa sì che i cambiamenti siano *catturati* sullo storage primario e solo in secondo momento propagati. Permette quindi di avere dei bassi costi, una grande scalabilità e flessibilità ma si rischia una perdita di dati o una perdita di consistenza. Viene usata per il load balancing e per quelle applicazioni dove è importante l'efficienza d'accesso.

#### CONTESTI CHE MOTIVANO LA REPLICAZIONE

Alcuni contesti sono la condivisione di dati tra utenti scollegati *(ad esempio un commesso viaggiatore che si replica sul proprio dispositivo mobile senza connessione il catalogo della propria compagnia)*, la data consolidation *(ad esempio una compagnia che ha dei dati in siti diversi quali negozi, siti di fabbricazione, siti di assicurazione; la replicazione può copiare i dati da ogni sito in uno centrale per analisi, report e data warehouse)*, la data distribution *(utile quando c'è un flusso bidirezionale di update)*.
Esistono anche aspetti legati alla performance, in particolare all'efficienza d'accesso *(spostare i dati vicino alle applicazioni)*, il load balancing e la disponibilità dei dati offline.

Chiaramente se l'applicazione in questione non ha bisogno di update immediati, si può usare la replicazione per ridurre il carico sul server principale.
Inoltre, se viene usata la replicazione per aumentare la disponibilità dei dati *(sistemi disaster - recovery)*, queste soluzione devono essere ampiamente testate e pianificate poiché non possono permettersi di fallire.

Esistono anche contesti di separazione tra data entry e report; un database che viene usato per molte operazioni di write *(data entry)* e molte operazioni di read *(report)* non sempre funziona in modo efficiente poiché può succedere che le transazioni che operano su quel database blocchino le tabelle che devono essere lette, rallentando il tutto. In questi casi, pur esistendo metodi per evitare i blocchi, è meglio separare i database di data entry e di report. Per quanto riguarda le transazioni questo

approccio funziona bene poiché gli update vengono trasferiti dal server di data entry a quello di report.

Un aspetto più complesso che richiede la replicazioni invece è la **coesistenza di applicazioni**. Potrebbero esistere delle trasformazioni complesse di dati richieste per accomodare nuovi requirments di una applicazione *(ad esempio quando esce una nuova versione di un'applicazione già in uso)*. I dati replicati devono quindi essere filtrati e/o trasformati in modo adeguato; è quindi necessario far coesistere il vecchio database con il nuovo il che spesso è molto complesso. A volte si fanno anche convivere le applicazioni vecchia e nuova effettuando migrazioni parziali.

#### CONTESTI IN CUI NON ANDREBBE USATA LA REPLICAZIONE

Anche se la replicazione di database ha molti benefit e può risolvere molti problemi, in alcune situazioni non è così ideale, ad esempio

- Quando ci sono **update frequenti** a molteplici repliche di record esistenti.
  Saranno di più i conflitti che le soluzioni, il che porterà all'inserimento di nuovi record nel database.
  Chiaramente questo implica la necessità di una risoluzione dei conflitti manuale che richiede molto tempo.
- Quando la **data consistency** è criticale sempre.
  Le applicazioni che hanno bisogno che le informazioni siano corrette sempre *(banche, biglietterie aeree, tracking)* usano delle transazioni ACID che richiedono l'aggiornamento istantaneo delle repliche *(come ROWA)* che porta ad una grossa perdita di performance.

## TIPOLOGIE DI REPLICHE

Esistono infine diverse tipologie di repliche.

- **One : Many** *(Data Distribution)*: la sorgente distribuisce in maniera sincrona o asincrona le varie copie. Le copie sono passive e vedono soltanto i dati *(es. data entry e reporting)*.
- **Peer - to - Peer**: in questa soluzione esistono vari nodi sorgente che possono scrivere e leggere e tutti i nodi si aggiornano tra loro. Gestibile con un approccio *ROWA*.
- **Many : One** *(Data Consolidation)*: abbiamo delle sorgenti locali *(ad esempio filiali di un negozio)* che trasferiscono i dati che vengono consolidati a livello centrale.
- **Bi - Directional**: sia la copia primaria che quella secondaria possono scrivere. Versione semplificata del Peer - to - Peer.
- **Multi - Tier Staging**: si hanno dei livelli intermedi. Usata ad esempio per soluzioni di *publish / subscribe* dove chi scrive deposita i cambiamenti nel livello intermedio da dove poi chi legge potrà andare a vedere se ci sono novità.

#### COME REALIZZARE UNA REPLICA

Esistono vari modi per creare delle repliche

- **Fisica**: in cui fisicamente un dipendente stacca la memoria di massa da un server, la copia in un altro e poi la riattacca al primo.
- **Backup**: invece di prendere la memoria di massa principale, si prende una memoria di massa di backup per la copia.
- **Replica Incrementale**: si fa un backup e, successivamente, si spostano solo i log, se servono dei dati si rieseguono le operazioni loggate. Alcune soluzioni usano delle code di *publish / subscribe*

per trasferire solo i log delle transazioni commitate ai server di replica che rieseguiranno le operazioni necessarie.

# NOSQL

Vedremo che esistono diversi modelli di rappresentazione dei dati non sequenziali come i modelli document based, i modelli a grafo, a chiave/valore e wide column.
Negli anni 70 si cercava un modello dati adatto a grandi banche di dati che dovevano essere condivise *(Large Shared Data Banks)* e si puntava molto sulla riduzione della ridondanza *(usando ad esempio tabelle di join)* poiché la memoria fisica costava molto: nacque così il modello relazionale.
Al giorno d'oggi chiaramente i prezzi sono molto inferiori quindi uno dei motivi che portò alla creazione dei modelli relazionali non è più valido e l'aspetto fondamentale è il tempo di risposta dei db.

Ma quali sono gli aspetti positivi del modello relazionale?

- **Modello molto ben definito**
  Utilizza il principio della minimizzazione e della *closed world assumption* che stabilisce che tutto quello che serve all'applicazione è presente all'interno del db.
  Chiaramente al giorno d'oggi l'assunzione del *closed world* non esiste più, solitamente un'azienda ha diversi dati dei propri utenti *(ad esempio dati provenienti dai social)*.
- **Tecnologia consolidata**
  Essendo una tecnologia nata negli anni 70, ha più di 35 anni di ricerca e sviluppo per quanto riguarda sicurezza, ottimizzazione e standardizzazione della tecnologia stessa.
- **Molti dati sono ancora immagazzinati in ambienti DBMS**
  E fare un porting dei dati da un modello ad un altro è un'operazione complessa e molto rischiosa.
- **Per molte applicazioni è ancora la scelta ideale**

Vediamo però anche i possibili aspetti negativi:

- **Modello molto ben definito**
  Quello che è un punto di forza può anche essere un punto debole per il modello relazionale poiché essendo così ben definito è anche molto restrittivo.
  Ad esempio abbiamo per ogni attributo un solo valore, non possiamo avere più valori per un attributo, per poter avere più valori associati ad un attributo bisogna avere un'altra tabella che però richiederà un join per raggruppare i dati, operazione costosa in termini di tempo.

  Inoltre, il modello relazionale non è del tutto compatibile col modello ad oggetti, difatti spesso vengono usati framework che facciano da middleware tra i due mondi.
- **I database relazionali sono difficili da modificare**
  È difficile modificare uno schema già implementato e spesso non è molto scalabile.

Inoltre, i database relazionali gestiscono bene lo *scale up*, ma hanno problemi con lo *scale out*.

- **Scale Up**
  Rappresenta una scalabilità verticale in cui si passa ad esempio da una macchina meno potente ad una più potente. Chiaramente questo tipo di scalabilità ha un limite oltre il quale non si può andare. Se serve un miglioramento anche leggero, viene imposto un upgrade.
- **Scale Out**
  Rappresenta una scalabilità orizzontale in cui ad esempio si aggiorna il sistema corrente. Un esempio sono i server che utilizzano delle *lame* di dischi; quando una di queste si riempie se ne

aggiunge un'altra. È la soluzione classica di utilizzo efficiente del cloud. Se serve un miglioramento leggero è possibile ottenerlo senza spreco di risorse.

Chiaramente per progetti in cui l'utilizzo delle macchine non è di lunga durata si può affittare una macchina potente e sfruttarla, mentre per progetti che richiedono ad esempio computazione h24 è più sostenibile usare un'architettura a microservizi con tante macchine di fascia inferiore.

Il termine NoSQL venne introdotto per la prima volta da Carlo Strozzi che sviluppò una API Linux per accedere a dati sequenziali senza usare il SQL: il significato del termine era *Non voglio usare SQL*.
Dai primi anni 2000 sono stati introdotti vari database con vari paradigmi di rappresentazione dei dati come i graph database, BigTable,  i document - based db. Nel 2009 venne quindi reintrodotto il termine NoSQL stando a significare questa volta *Not Only SQL*.

Dunque, *Not Only SQL* sta a rappresentare un insieme di modelli di rappresentazione dei dati e relativi software di gestione. Questi modelli hanno tre caratteristiche fondamentali:

1. **Schema free**
   Tutti questi modelli sono *schema free* o *schemaless*.
   Nei modelli relazionali solitamente prima si definisce il modello *(gli attributi che descrivono i dati e le loro relazioni)* e poi si popola il db. Se c'è il bisogno di aggiungere un nuovo attributo o di modificarne uno esistente, prima si modifica il modello e poi *(se possibile)* si modificano i dati.

   Nella maggior parte dei modelli NoSQL non c'è un modello fisso, ma questo è basato sui dati inseriti. Questo permette a questi modelli di usare la *open world assumption*.
   Ovviamente tutto ciò può creare problemi di consistenza dei dati ma allo stesso tempo permette di avere una grande libertà di programmazione.
2. **Usano il CAP Theorem**
   Il CAP Theorem stabilisce che un sistema distribuito non potrà mai fornire simultaneamente tutte e tre le seguenti caratteristiche: **C**onsistency *(tutti i nodi vedono gli stessi dati allo stesso tempo)*, **A**vailability *(una garanzia che ogni richiesta riceverà una risposta, di successo o meno)*, **P**artition tolerance *(il sistema continua ad operare nonostante arbitrarie perdite di messaggi o fallimenti di parti del sistema)*.

   Un sistema distribuito potrà fornire solo due delle caratteristiche allo stesso tempo.
   Gli RDBMS di solito sono CA, eventualmente con un numero limitato di nodi si può avere un CAP.
   I sistemi NoSQL sono CA *(i dati sono coerenti ma non sempre il dbms può funzionare 24/7)* o AP *(a volte i dati potrebbero non essere consistenti)*.
3. **Modello BASE *(Basic Availability, Soft state, Eventual consistency)***
   Basic Availability significa sostanzialmente dare sempre una risposta, anche con una consistenza parziale.
   Soft State sta ad indicare l'abbandono dei requirment di consistenza del modello ACID.
   Eventual Consistency indica che eventualmente, in futuro, i dati convergeranno in uno stato consistente. Abbiamo quindi una consistenza ritardata a differenza di quella immediata del modello ACID.

Alcune tipologie di database NoSQL sono:

- **Key - Value**: tabelle di hash dove la chiave punta ad un particolare valore. Il mapping tra chaivi e valori e realizzato mediante hashing per massimizzare le performance.
- **Wide Column** dove la chiave punta a colonne multiple.
- **Document Based** dove la chiave indirizza dei documenti *(ad esempio oggetti JSON)*.
- **Graph DB**: costituiti da nodi e relazioni tra nodi *(archi)*. I nodi hanno delle proprietà, che sono delle informazioni pertinenti ai nodi. I graph db in generale non scalano bene.

Una differenza fondamentale rispetto al modello relazionale che mette tutti i dati allo stesso livello di importanza, i modelli NoSQL mettono un concetto ad un livello di importanza maggiore *(la root ha maggiore importanza)*. Pensiamo ad esempio a due tabelle *Person* e *Car* e ad un documento JSON associato ad una persona con una chiave *cars* contenente i documenti delle macchine appartenenti ad essa *(le relazioni sono incluse nei dati)*. Per vedere quante macchine ha una persona nel modello NoSQL basta una lettura lineare, mentre nel modello relazionale va fatto un join, per vedere di chi è una macchina bisogna leggere tutti i documenti *persona* e i relativi campi *cars*, mentre per il modello relazionale si fa ancora un join.

## SISTEMI DOCUMENT - BASED

Come detto in precedenza, nei sistemi document - based solitamente i dati sono rappresentati come documenti *(ad esempio oggetti JSON)* e sono indicizzati da chiavi uniche.

Il modello documentale, a differenza di quello relazionale, è un albero; se vogliamo accedere a delle informazioni che si trovano nei nodi foglia dobbiamo percorrere tutto l'albero, mentre possiamo leggere subito la radice. Come detto prima, questo significa dare una maggiore importanza a certe letture *(se vogliamo leggere le foglie di una radice basta percorrere l'albero, mentre se vogliamo sapere a quale radice appartiene una certa foglia dobbiamo leggere tutti gli alberi)* mentre nei modelli relazionali tutte le letture hanno uguale importanza *(si fanno sempre dei join)*.
Generalmente questo modo di rappresentare i dati si avvicina maggiormente a come gestiamo manualmente i dati solitamente.

Questo modello è multidimensionale il che significa che ogni nodo può contenere 0, 1 o più valori *(altri documenti)* e le interrogazioni si possono fare a qualsiasi livello su qualsiasi campo. Chiaramente lo schema è flessibile e si adatta ai dati: questo implica anche che le interrogazioni fatte a questi sistemi dovranno infierire lo schema.
Inoltre, avere le relazioni rappresentate direttamente nei dati permette una località maggiore dei dati, richiede meno indici e di conseguenza permette di avere delle performance migliori.

#### REFERENCING VS EMBEDDING

Nel modello *document - based* si possono rappresentare dei concetti sia facendo embedding delle informazioni *(rappresento le relazioni direttamente nei dati, ho dei documenti nei documenti)* che facendo un *referencing* usando degli id come nel modello relazionale *(utile ad esempio se più documenti devono contenere uno stesso altro documento, ad esempio una famiglia che condivide lo stesso indirizzo)*.
Ovviamente l'embedding rischia di aumentare la replicazione dei dati portando a problemi di qualità dei dati e di spazio. Abbiamo però detto che al giorno d'oggi la quantità di spazio non è più un problema così grande, dunque, in un'applicazione che fa accesso frequente a questo tipo di dati può aver senso utilizzare l'embedding per massimizzare i tempi di risposta.

#### LINGUAGGIO DI INTERROGAZIONE

Abbiamo visto come il modello relazionale abbia una sorta di standard per le interrogazioni, ovvero il SQL, mentre per i modelli *document - based* esistono vari linguaggi, non c'è n'è uno singolo.

## MONGODB

Un sistema *document - based* è MongoDB che memorizza i dati in formato **BSON *(Binary JSON)*** e accede ad essi tramite degli indici. Supporta il referencing dei dati ma chiaramente è più veloce usare un approccio embedded evitando join.

Quelle che per un database SQL sono tabelle, per i database *document - based* sono **collezioni**.
Le righe delle tabelle sono i **documenti** e le colonne sono le **chiavi**.
Tendenzialmente i join sono rappresentati dai **documenti embedded** e le chiavi esterne dai **reference**.

L'architettura di MongoDB prevede alcuni vari engine di basso livello, componenti di sicurezza e di management e il **MongoDB Document Data Model** e il **MongoDB Query Language *(MQL)*** proprietario che è quello che viene usato per interrogare un database MongoDB.
Uno dei punti forte di MongoDB è che il suo linguaggio di interrogazione è molto simile ai linguaggi di programmazione *(specialmente ad oggetti)*, cosa che ne facilita l'uso ed evita l'utilizzo di middleware per l'object persistencing.

Il linguaggio di MongoDB permette anche di creare degli indici rapidamente, utili a minimizzare i tempi di risposta, nel seguente modo
$\texttt{db.collection.ensureIndex}(\{\texttt{'author'} : 1\})$
Dove $1$ significa creare un indice crescente, mentre $-1$ ne fa creare uno discendente.

Gli indici si possono creare a qualsiasi livello del documento
$\texttt{db.collection.ensureIndex}(\{\texttt{'comments.author'} : 1\})$

Esistono anche indici geospaziali
$\texttt{db.collection.ensureIndex}(\{\texttt{'author.location'} :\texttt{'2d'}\})$

#### REPLICAZIONE

MongoDB rientra nella categoria CP del CAP Theorem, quindi garantisce consistenza e partition tollerance.
Crea le sue repliche usando architetture *master / slave (o primary / secondary)* dove le scritture avvengono sul *primary* che poi si propagheranno ai *secondary*.

In particolare, MongoDB utilizza una replica parziale incrementale dove all'inizio viene fatto un *sync* tra *primary* e *secondary* e poi l'aggiornamento viene fatto guardando i file di log. La garanzia di tolleranza viene garantita tramite un a elezione.

Un nodo può assumere uno dei seguenti stati:

- **STARTUP**: un nodo che non è ancora membro di nessun set di replica.
- **PRIMARY**: l'unico nodo nel set di replica che accetterà le operazioni di scrittura.
- **SECONDARY**: un nodo che effettua solo la replicazione dei dati *(può essere promosso a primary)*.
- **RECOVERING**: un nodo che sta effettuando una qualche operazione di recupero dei dati *(può essere promosso a primary)*.
- **STARTUP2**: un nodo appena entrato nel set *(può essere promosso a primary)*.
- **ARBITER**: un nodo non atto alla replicazione dei dati ma con lo scopo di prendere parte alle elezioni per promuovere i nodi a primary *(può essere promosso a primary)*.
- **DOWN / OFFLINE**: un nodo irragiungibile.
- **ROLLBACK**: un nodo che sta svolgendo un *rollback* e quindi sarà inutilizzabile per le letture *(può essere promosso a primary)*.

Se il nodo *primary* risulta *down* alle altre repliche si effettua una elezione e si sceglie il nodo con la versione più aggiornata dei dati. Durante le elezioni le scritture si interrompono.

**ELEZIONI**
Ogni nodo nel set di replica manda un *heartbeat* agli altri nodi *(con timeout)* e ogni nodo ha un identificativo che rappresenta la sua priorità durante l'elezione *(più alto è più è probabile che venga scelto)*.
Esiste una gerarchia tra nodi in base al loro livello di priorità in modo che appena il *primary* diventa *down*, il *secondary* col maggior livello di priorità richiederà delle nuove elezioni.

Un set di replica può avere al massimo 50 membri di cui solo 7 votanti *(i non votanti hanno priorità pari a 0)* e i nodi votanti daranno al più un voto durante le elezioni. Un nodo con priorità 0 può comunque mantenere una copia dei dati, accettare le letture e votare nelle elezioni.

**POLITICHE PER LA SCRITTURA**
Durante un'operazione di scrittura è possibile impostare un'opzione `writeConcern` per indicare al sistema il comportamento per le repliche che prende i seguenti parametri:

- **w**: il numero di nodi in cui il dato deve essere replicato prima che l'operazione possa essere considerata conclusa.
- **j**: l'intenzione di scrivere sui log di *Wired Tiger (uno degli engine)* PRIMA che l'operazione sia stata eseguita *(write ahead logging)*.
- **wtimeout**: il tempo limite *(ms)* da aspettare nel caso in cui *w* sia maggiore di 0. Chiaramente se *w* = 0 non bisogna attendere prima di concludere l'operazione.

In caso di grandi quantità di dati da scrivere solitamente conviene evitare *w* troppo grandi ed evitare il *write ahead logging* per minimizzare i tempi.

**FRAMMENTAZIONE**
MongoDB garantisce un meccanismo di *sharding*, ovvero di scalabilità orizzontale dei propri elementi che viene effettuato sulla base della chiave dell'oggetto *(hash - based)*, sulla base di un range *(range - based)* o sulla base di alcuni valori assunti da attributi particolari *(tag - based)*. Inoltre, MongoDB offre un bilanciamento automatico dei shard.
Si utilizzano delle *chiavi di shard* suddivise in *chunk* e il numero massimo di *chunk* che è possibile definire su una sharded key rappresenta il numero massimo di shard in un sistema.

Per la frammentazione esistono tre ruoli:

- **Router *(Mongos)***: l'entry point del cluster, tutte le query verranno eseguite su questo processo. Dopo aver visto su quali shard sono i dati grazie al config server sceglierà se fare una target query *(i dati sono solo su uno shard)* o una broadcast query *(dati su più shard)*.
- **Shard**: istanza di MongoDB.
- **Config Server**: server che ospitano file di configurazione del cluster, necessari al sistema per avere informazioni sulla posizione dei documenti nei vari shard.

MongoDB richiede che sia gli shard che i config server siano configurati su dei replica set in modo da garantire la disponibilità in caso di guasti e non avere un unico *point of failure*.

Esistono dei *primary shard* che contengono tutto il database e gli *shard secondari* che contengono frammenti e repliche di alcune collezioni.

**POLITICHE PER LA LETTURA**
Esistono anche delle politiche per la lettura espresse tramite `readConcern` che può avere i seguenti valori

- **Local**: valore di default, dice al nodo di fare la query localmente. La lettura viene effettuata senza verificare che il dato sia stato scritto sul resto dei nodi. Nel caso di un *primary shard* non succede nulla di particolare, nel caso di *secondary shard* e i dati richiesti non sono su di esso, local va a fare la query sul nodo più vicino.
- **Available**: default per leggere i dati dai nodi secondari.
- **Majority**: leggo quando la metà più uno delle repliche ha ricevuto un ack di scrittura sul dato.

Chiaramente la scelta delle politiche di scrittura e lettura dipendono anche dal carico applicativo; se ho un'applicazione che fa tante letture ma poche scritture può aver senso usare una *majority* in scrittura e una *local* in lettura.

**TRANSAZIONI**

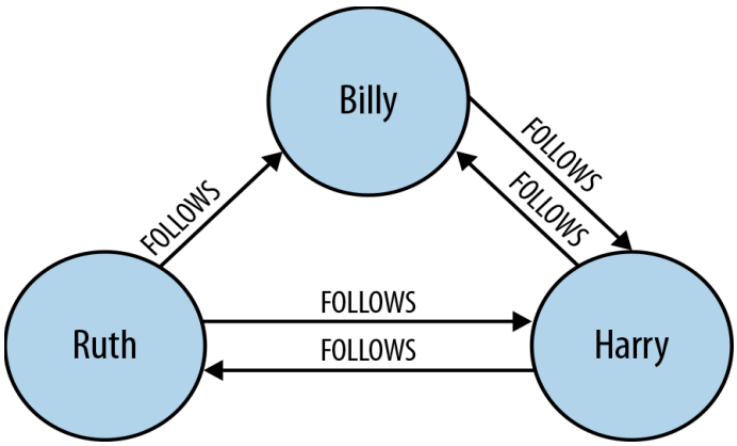
Dalla versione 4.0 MongoDB gestisce anche le transazioni usando l'engine *Wried Tiger* tramite l'utilizzo di *lock*. Esistono vari tipi di *lock*:

- **S**: Lock condiviso *(Shared)*, utilizzato per le letture.
- **X**: Lock di tipo esclusivo *(eXclusive)*, usato per le scritture.
- **IS**: Intenzione di bloccare in modo condiviso uno dei nodi che discende dal nodo corrente *(Intent Shared)*.
- **IX**: Intenzione di bloccare in modo esclusivo uno dei nodi che discende dal nodo corrente *(Intent eXclusive)*.

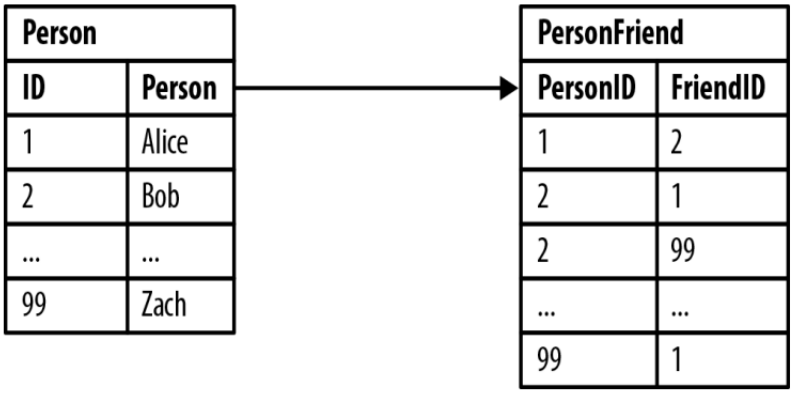
Essendo Wired Tiger un modello distribuito, anche le transazioni funzionano solo se i dati sono distribuiti. La motivazione è che avendo un solo nodo probabilmente si ha una sola applicazione che vi accede.

**GRAPH DATABASE**

Sappiamo già che un grafo $G = (V, E)$ è una collezione di nodi e archi, ovvero è un'insieme di nodi e di relazioni fra nodi. Inoltre, poiché esiste già una grandissima letteratura legata ai grafi, questa può essere riapplicata ai graph db.
Nei graph database i concetti più importanti *(l'equivalente delle entità in ER)* sono modellati come nodi mentre il modo in cui sono semanticamente associati si modella con le relazioni. Tra le applicazioni più gettonate per i graph db abbiamo i social, i sistemi di raccomandazione, di logistica, di bionformatica, identificazione di frodi.
Chiaramente la semantica di un graph db viene data dalle sue etichette; con i nodi rappresentiamo concetti diversi e con gli archi relazioni diverse che sono diversificati dalle etichette.



Pensiamo ora ad un semplice database che rappresenta un gruppo di amici e vediamo come rappresentarlo in vari paradigmi iniziando da quello relazionale.



| Person | |
|---|---|
| ID | Person |
| 1 | Alice |
| 2 | Bob |
| ... | ... |
| 99 | Zach |

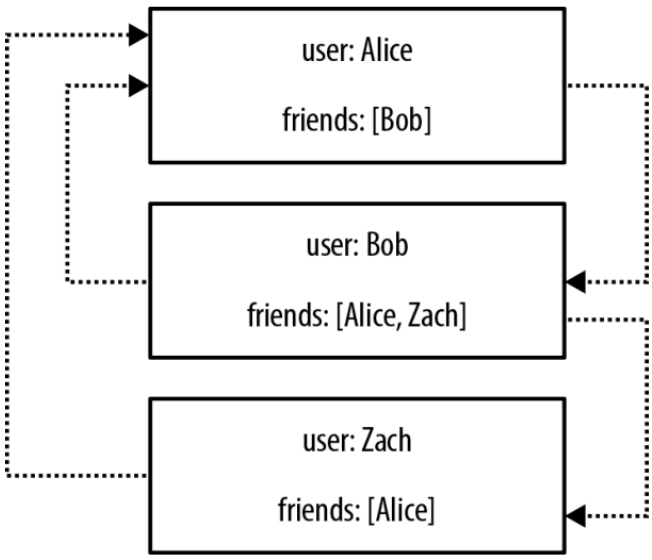
| PersonFriend | |
|---|---|
| PersonID | FriendID |
| 1 | 2 |
| 2 | 1 |
| 2 | 99 |
| ... | ... |
| 99 | 1 |

Supponiamo ora di voler sapere quali sono gli amici degli amici di Alice: dovremmo andare a vedere in *PersonFriend* chi sono gli amici di Alice, ovvero a chi corrisponde il *FriendID* quando il *PersonID* è 1 e poi, trovati questi, usare loro come *PersonID* per trovare i rispettivi amici degli amici.

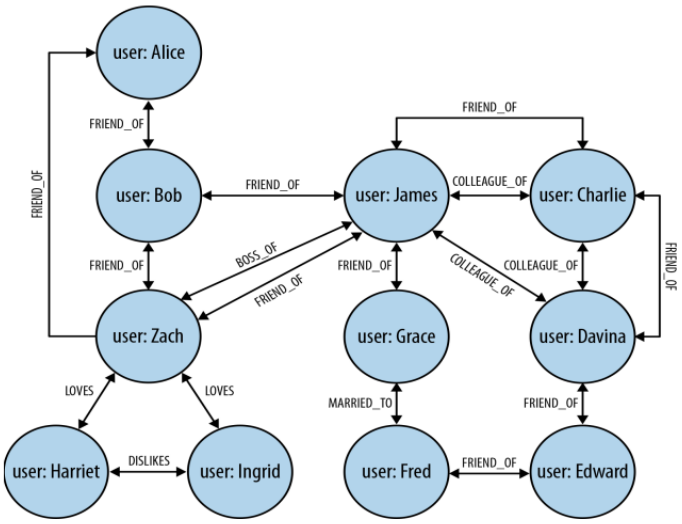
```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
  ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2
  ON pf2.PersonID = pf1.FriendID
JOIN Person p2
  ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

Chiaramente questa operazione non ha la migliore delle tempistiche dati tutti i join presenti.

Se rappresentassimo questa cosa in un sistema *document - based* avremmo due scelte: usare il referencing o l'embedding. Usando il referencing è come usare le chiavi esterne dei RDBMS, mentre usando l'embedding dobbiamo definire una gerarchia *(un albero)* e abbiamo quindi una modellazione più complicata e poco adatta al caso presentato.




Nel modello a grafo possiamo invece rappresentare direttamente le relazioni tra i vari nodi senza preoccuparci di gerarchie o tabelle di join. Notiamo che tra due nodi possiamo avere più relazioni. Diciamo che i graph db abbraccia le relazioni, cosa che lo rende molto intuitivo per noi umani, difatti la modellazione è molto simile a quella umana.




I nodi inoltre possono avere più attributi *(in questo caso hanno solo user)* che ne descrivano le caratteristiche proprie. Anche qui si ha la libertà dello schema e quindi ogni nodo può avere qualsiasi set di attributi.
Nei *Property Graph* anche le relazioni possono avere degli attributi. Gli attributi solitamente sono del tipo *chiave / valore* o *chiave / array di valori*.

Un problema che può sorgere è che possiamo rappresentare un concetto sia come come una relazione con un nodo a parte che come una proprietà del nodo attuale *(pensiamo ad esempio ad un indirizzo di consegna per un ordine)*.

Come nel caso dei *document - based*, anche per i graph db abbiamo vari linguaggi di interrogazione che dipendono dal dbms in sé. Un ulteriore cambiamento è che, mentre il modello relazionale immagazzina i dati secondo uno certo schema logico e i sistemi document - based in json binari, i graph db possono immagazzinare i dati internamente sia come dei grafi che in altri modi *(ad esempio con una tabella 'nodi' e una 'archi')* e decidere di processare i dati in un modo ortogonale a come sono stati immagazzinati.

Esistono due approcci per gestire un grafo:

- **Graph Database**: Un DBMS che gestisce in maniera persistente un grafo e usato per le transazioni *(OLTP)*. Permette chiaramente di eseguire operazioni CRUD ed è usato per dati complessi e connessi.
  Le interrogazioni sono risolte tramite **attraversamenti di grafi** che concettualmente corrispondono ai join dei modelli relazionali ma sono più efficienti in termini di performance.

  Difatti dove nei RDBMS abbiamo un join bomb, nei graph db ci basta seguire i riferimenti di ogni nodo *(pensando ancora all'esempio degli amici degli amici)*.
- **Graph Compute Engines**: Tecnologie per l'analisi off - line dei grafi *(OLAP)*.

Possiamo poi avere sistemi nativi e non per lo storage e per il processamento dei dati; chiaramente quelli nativi sono ottimizzati per la progettazione e gestione dei grafi *(usando ad esempio l'index free adjacency che scrive i nodi connessi vicini nei file di dati, molto efficiente in lettura ma meno in scrittura)*, mentre quelli non nativi trasformano i dati dal formato *grafo* ai formati relazionale, object oriented o altro.
I sistemi non nativi vengono usati principalmente perché i grafi non scalano bene orizzontalmente *(a meno che non abbiamo dei graffi disconnessi)* e quindi, per supportare la scalabilità e la frammentazione, è necessario utilizzare dei sistemi che le supportino. Chiaramente questo necessita anche di uno stato software che traduca le interrogazioni dal modello a grafo al modello non a grafo.

**CYPHER**
Uno dei linguaggi di interrogazione per i graph db è *cypher* ed è un linguaggio di pattern - matching di tipo dichiarativo. Cerca di avvicinarsi al linguaggio umano e di essere espressivo.

La rappresentazione è del tipo:
`(nodo1) − [: etichetta arco] − > (nodo2)` *(notare la direzione dell'arco)*
Mettendo più condizioni separate dalla virgola equivale a considerarle in and.

Un esempio di interrogazione in Cypher è

```
MATCH(c : user)
WHERE (c) − [: knows] − > (b) − [: knows] − > (a), (c) − [: knows] − > (a), c. user =′ Michael′

RETURN a, b
```

Che equivale al trovare tutti i nodi `c` con la label `user` tali per cui `c` conosce qualcuno che a sua volta conosce qualcuno e che `c` conosca anche questo secondo qualcuno e tali per cui `c` ha la label `user` pari a *'Michael'*. Ritorna poi i due *qualcuno*.
Possiamo generalizzare questa query come 'trova tutti gli utenti che conoscono Michael e almeno un altro amico che conosce anche Michael'.

**NOTA**: Notiamo che il valore di ritorno della query è una tabella. Difatti, non sempre un'interrogazione su un grafo restituisce un grafo come risposta a differenza di come fanno i document - based e i modelli relazionali.

## KEY VALUE

Il modello *key / value* è il più semplice modello NoSQL. Si possono associare dei valori alle chiavi e leggere tali valori conoscendone la chiave. È possibile anche associare ai valori dei tipi.
Chiaramente permette un accesso veloce tramite meccanismi di hash. Solitamente questi sistemi vengono utilizzati in memory.

## WIDE COLUMN

Anche se storicamente non è così, possiamo pensare al modello *wide column* come ad un evoluzione del modello *key / value*: il concetto è che quello che indirizza la chiave non è più un valore unico ma una *riga* che contiene degli attributi monovalore. Diciamo che è una via di mezzo tra il *key / value* e il *documentale*.

**BIG TABLE**
Un esempio di questi sistemi è *Google Big Table* che sfrutta una mappa multidimensionale ordinata, persistente e sparsa. La mappa è indicizzata tramite una ***row key***, una ***column key*** e un ***timestamp***. Questo permette di avere, per le stesse righe e colonne, valori diversi in base al loro timestamp.

Si usa sempre un approccio *key / value* dunque non è facile fare range query *(trova tutti gli studenti il cui nome inizia con 'a')*, ma permette un veloce lookup.

In generale la tabella è organizzata in **Column Family** che ha un nome e contiene una o più colonne che hanno una certa relazione tra loro. Le colonne infine appartengono ad una sola Column Family e sono incluse nella riga. Le colonne in sé non fanno parte dello schema *(e sono quindi dinamiche)* che è caratterizzato solo dalle Column Families.

Il paradigma wide column tende a denormalizzare i dati e a replicarli piuttosto che a richiedere join per risparmiare tempo. Inoltre, il timestamp permette anche di avere un versioning dei dati.

Una versione distribuita di Big Table è HBase che utilizza un'architettura master / slave dove il master governa una *HRegionServer* con tanti region server, dove la *region* è un sottoinsieme di righe di una tabella.
I *Region Server* gestiscono le varie regioni di dati e permettono letture e scritture *(sincronizzate tramite file di log)*. Il master coordina i slave, assegna regioni e rileva failures.

**CASSANDRA**
Cassandra, che è un'evoluzione open source del database NoSQL di facebook, è un modello essenzialmente *chiave / valore* che utilizza però anche concetti derivati dai *wide column*.
Cassandra utilizza il *Cassandra Query Language (CQL)* che è molto simile al SQL.

Cassandra utilizza inoltre un'architettura distribuita *peer - to - peer* dove viene distribuito lo spazio delle chiavi. I nodi sono organizzati secondo una topologia ad anello. Solitamente la replicazione avviene anche sul nodo successivo e sul successivo del successivo.
Per rilevare failures vengono usati dei *protocolli di gossip* dove ogni nodo chiede al proprio successivo se va tutto bene.

Cassandra, come MongoDB, permette di selezionare delle politiche per la scrittura e la lettura dei dati che permette di avere una consistenza dei dati decidendo su quali operazioni porre maggior carico applicativo.
Possiamo ad esempio dire che una lettura è certa se risponde un solo nodo oppure un certo numero di nodi, idem per le scritture.

Quando i dati subiscono una delete vengono solo marchiati per la cancellazione poiché è più veloce e successivamente verranno cancellati in massa.